Implementing a GDB Stub in Lightweight Kitten OS

Angen Zheng, Jack Lange
Department of Computer Science
University of Pittsburgh
{anz28, jacklange}@cs.pitt.edu

ABSTRACT

Because of the increasing complexity of the applications running in Kitten, a lightweight HPC OS targeted for compute nodes of massively-parallel, distributed-memory supercomputers, and the complex hardware that Kitten is running on, bugs are becoming more difficult to find. As a result, the need for Kitten to support user-level application debugging becomes more critical. Unfortunately, Kitten currently has no support for user-level application debugging. To solve this problem, we implemented a GDB stub as a kernel module for Kitten without changing the internal architecture of Kitten and Palacios, a high performance embeddable VMM for Kitten, yet with minimal modification to Kitten and Palacios codebases.

The GDB stub has only 590 lines of changes to Kitten and 238 lines of changes to Palacios in total. Along with the GDB stub and Palacios, we are now able to debug a user-level application running in Kitten from a GDB client running in a Palacios Linux Guest. This paper presents our design and implementation of the GDB stub in Kitten.

1. INTRODUCTION

Palacios [1] and Kitten [2, 3] are open source tools that enable applications to obtain scalable high performance on large machines. Kitten is an open-source, lightweight kernel OS developed at Sandia National Laboratories, specialized for high performance computing (HPC). Palacios is an open-source, high performance, type-I pure virtual machine monitor (VMM) making extensive, non-optional use of hardware virtualization technologies. As part of the V3VEE project (http://v3vee.org), Palacios is designed to be an OS independent VMM, which can be embeddable into different host OSes, such as Linux and Kitten. Together, Palacios and Kitten provide a thin layer over the hardware to support both full-featured virtualized environments and native codebases, which broadens the applicability and usability of HPC systems.

However, due to the increasing complexity of the applications running in Kitten and the accelerated hardware that Kitten is running on, bugs are becoming more difficult to find, making the need for Kitten to support user-level application debugging becomes evident. Unfortunately, Kitten currently has no support for user-level application debugging. To solve this problem, we implemented a GDB stub as a kernel module for Kitten without changing Kitten and Palacios architecture yet with minimal modification to Kitten and Palacios codebases. The GDB stub has only 590 lines of changes to Kitten and 238 lines of changes to Palacios in total. Along with the GDB stub and Palacios, we are now able to debug a user-level application running in Kitten from a GDB client running in a Palacios Linux Guest.

Adding debugging capability to Kitten is not an easy task. First, we could not change Kitten and Palacios architecture yet we need to minimize modification to Kitten and Palacios codebases. Secondly, unlike Linux, Kitten doesn't support ptrace interface and doesn't fully support signals at present. This not only prohibits GDB/GDBServer [4] from running natively on Kitten since both GDB and GDBServer rely on ptrace interface and signals, but also requires implementing the GDB stub from scratch, such as manually stop/resume a running process and do register/memory examination.

Fortunately, Kitten has implemented the KGDB stub, from which we derived our GDB stub. Nevertheless, the adaption of the KGDB stub is also not easy. Unlike KGDB [5, 6] running in the same address space with the kernel, the GDB stub is running in kernel space while the debugging process is running in userspace, requiring address translation from debugging process's address space to kernel address space for memory read/write. Besides, once KGDB takes the control over the kernel, it stalls all the other processors to simplify multi-threaded debugging, which is undesired by GDB stubs for user-level application debugging. Furthermore, KGDB only has one debugging target, the kernel, while the GDB stub needs to be able to debug multiple user-level processes at the same time.

In the remainder of this paper, we present the design and implementation of the GDB stub in Kitten. The core contributions of this paper are the following:

- We describe the design and implementation of the GDB stub for Kitten.
- We add the debugging facility to Kitten without chang-

Command	Description	RSP Packet Example
g	Read all registers	Request: $g#67 Reply: $reg0reg1reg2.....#xx
G	Write all registers	Request: $Greg0reg1reg2.....#kk Reply: $OK#9a
m	Read memory at address	Request: $m6be0cc,4#c0 (Read 4 bytes starting from address 0x6be0cc) Reply: $00000011#82
M	Write memory at address	Request: $M61be0cc,4:00000001#e6 (Write 4-byte 00000001 into address 0x6be0cc) Reply: $OK#9a
Z	Set breakpoint at address	Request: $Z0,401e38,1#78 (Set a breakpoint at address 0x401e38) Reply: $OK#9a
z	Remove breakpoint at address	Request: $z0,401e38,1#98 (Remove the breakpoint at address 0x401e38) Reply: $OK#9a
c	Continue the program	Request: $c#63 Reply: $T05#b8
s	Single –stepping the program	Request: $s#73 Reply: $T05#b8
?	Get last signal	Request: $?# Reply: $T05#b8 (The process stopped because of a breakpoint exception)
D	Detach from the target	Request: $D#44 Reply: $OK#9a

Figure 1: Minimal Essential RSP Commands

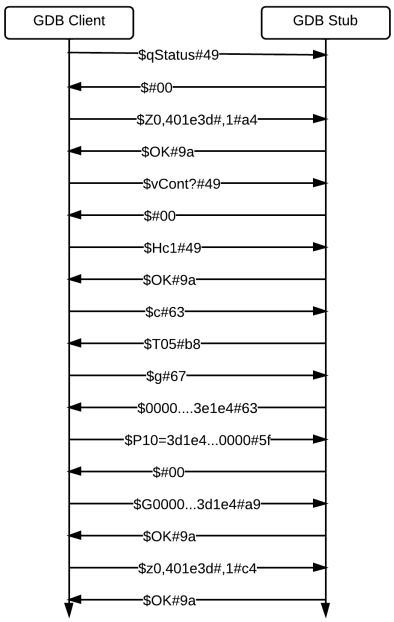

Figure 2: RSP Packet Exchanges for GDB continue Command

ing Kitten and Palacios architecture yet with minimal modification to Kitten and Palacios codebases. For Kitten, we add 30 newly lines to 5 existing C files and 5 new C files with 560 lines of code. For Palacios, we add 3 new C files with 238 lines of code.

2. RELATED WORK

A lot of work has been done on how to write a GDB stub for embedded systems [7, 8, 9] or specific hardware architectures [10]. Paper [7] gives a summary of the GDB Remote Serial Protocol (RSP) [11] with concrete examples. The protocol specifies how GDB communicates with any GDB stub. It supports various types of connection: serial devices, TCP/IP, UDP/IP, and pipes. Also, the protocol defines a minimal set of RSP commands that any GDB stub should implement. Figure 1 shows a subset of the essential RSP commands. All GDB commands like target remote, break, continue and step is actually mapped into a serial of RSP commands, which are sent to the GDB stub via RSP packets. RSP packets begin with a dollar sign ($), followed by the packet data, and end with a pound sign (#) and one byte packet data checksum. Both the packet data and the checksum are represented as ASCII hex characters. Figure 2 shows an example of RSP packet exchanges of GDB continue command.

Paper [7] presents an overview about how GDB works from an embedded perspective, especially in aspect of breakpoint and single-stepping implementation. Article [9] describes how to implement a minimal GDB stub in userspace based on ptrace interface for embedded systems. In paper [10], the author shows how to implement a GDB stub for OpenRISC 1000 architecture with concrete examples of the mapping between GDB commands and RSP commands.

3. KITTEN GDB STUB

3.1 Architecture

As identified by the double dash round box in Figure 3, the debugging facility consists of a GDB virtio console backend in Palacios and a GDB stub kernel module in Kitten.

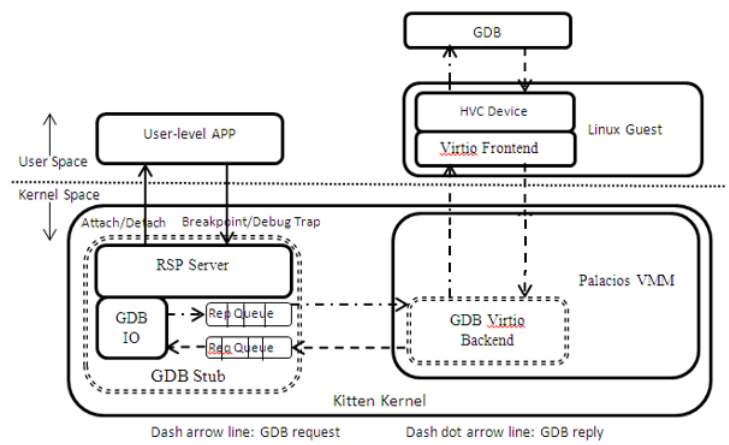

Figure 3: GDB Stub Architecture

3.1.1 GDB Virtio Console Backend

The virtio console frontend and backend along with the virtio console device driver in the guest OS provide a fully virtualized serial device for the guest OS, such as /dev/hvc0. As a result, the GDB client running in the Linux Guest can connect to the GDB stub running in Kitten via these serial devices. During the communication, the GDB virtio console backend simply forwards GDB requests to the GDB stub, then waits for GDB replies, and finally forwards GDB

replies to the virtio frontend, which, in turn, forwards GDB replies to the GDB client running in the Linux Guest.

3.1.2 GDB Stub

The GDB stub is an implementation of RSP. However, our current implementation only supports serial device communication, a subset of RSP commands, as well as all-stop multi-threaded debugging. In all-stop mode, all threads of the debugging process stop whenever any thread of interest stop either because of a breakpoint or single-stepping exception. This simplifies the implementation of multi-threaded debugging [12]. Conversely, all threads resumes whenever the thread of interest resumes. In other words, whenever the GDB stub takes over the control of the debugging process, only one thread is the focus of debugging. Therefore, the GDB stub needs to serialize concurrent breakpoint or single-stepping exceptions from different threads, ensuring that only one GDB stub instance is active for each debugging process at any point of time. That is, in cases where multiple threads of the same debugging process encounter a breakpoint or single-stepping exception at the same time, only one of them is allowed to be handled by the GDB stub. All other threads have to wait for the active GDB stub instance completes its handling.

To debug a user-level process running in Kitten, we first need to start the GDB stub in Kitten, which will pause the running process and wait for GDB clients to connect. Once the GDB client connects, the user is able to do register and memory examination, set breakpoint and single-stepping. In the following sections, we will describe how the GDB stub supports process stopping and resuming, register and memory examination, setting breakpoint as well as single-stepping.

3.2 Stop and Resume a Process

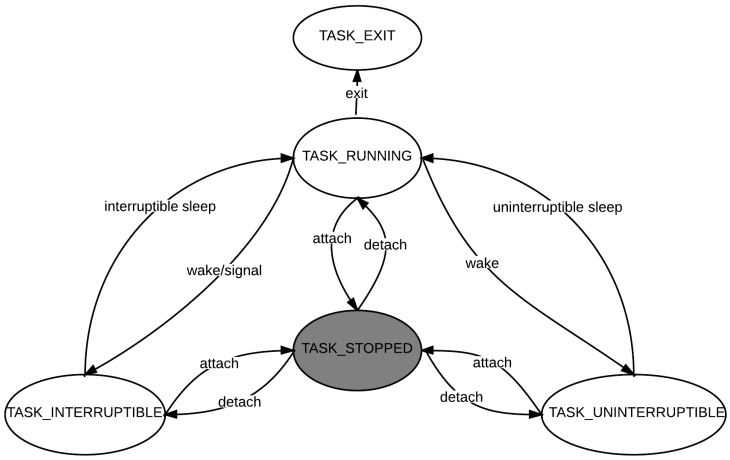

Figure 4: Process State Transitions

In Linux, whenever GDB attaches to a process, it sends a SIGNAL_STOP single to the process to stop it. Successively, GDB can resume the process by sending a SIGNAL_CONT single to the process. However, Kitten doesn't have full support for signals at present. Thus, we need to manually stop or resume a process. To minimize modifications to Kitten, we add a new process state TASK_STOPPED to Kitten. Originally, Kitten only has 4 process states: TASK_RUNNING, TASK_EXIT, TASK_INTERRUPTIBLE and TASK_UNINTERRUPTIBLE. Figure 4 shows the transitions between each state.

As a result, the GDB stub can now simply stop a running process by setting its state to TASK_STOPPED, and resume it by switching it back to TASK_RUNNING state with a reschedule on that CPU. The reason behind this is that for processes previously in TASK_RUNNING state, it will not be scheduled again unless we put it back to TASK_RUNNING state. However, for processes originally in TASK_(UN)INTERRUPTIBLE state, things become a bit more complicated. The GDB stub need to keep track of the events happening to wake up the process while it is in TASK_STOPPED state. Again, to minimize changes to Kitten, the GDB stub takes advantages of the unused ptrace flag of task_struct in Kitten to record the process's original state. Whenever an event happens to wake up the process while it is in TASK_STOPPED state, instead of updating the process state, the GDB stub updates the ptrace flag from TASK_(UN)INTERRUPTIBLE to TASK_RUNNING. Later on, the GDB stub can resume the process by setting its state to the value of ptrace flag. In this way, the GDB stub will not miss any wake up events, even though the process is in TASK_STOPPED state. For multi-threaded processes, we can apply the same algorithm above to stop or resume all threads because both processes and threads are represented with same data structure (task_struct) in Kitten.

3.3 Register Examination

The GDB client issues g/G command to the GDB stub whenever it needs to read/write the registers of the current debugging thread. Upon receiving a g command, the GDB stub simply packets all register values from the task_struct of the current debugging thread into a RSP packet in an order expected by GDB client. For a G command, the GDB stub simply updates the current debugging thread's registers with the values provided by GDB client.

3.4 Memory Examination

Whenever the GDB client wants to know/update the content of a specific memory address, such as print the value of a global variable or replace the opcode at the breakpoint address with a trap instruction, it will send m/M commands to the GDB stub. Unfortunately, the GDB stub could not read/write the memory address directly, because the debugging process and the GDB stub is running in different address spaces. As we can see from Fig. 1, the GDB stub is running as a kernel module while the debugging process is running in userspace. Therefore, to do memory examination, we first need to translate the memory address from the debugging proccess's address space to kernel address space. Fortunately, Kitten already implements the function of converting a memory address from one process's address space to another. Otherwise, we have to traverse through the debugging process's page table to figure out the corresponding physical address, and then covert it into kernel virtual address via __va() macro.

3.5 Breakpoint Management

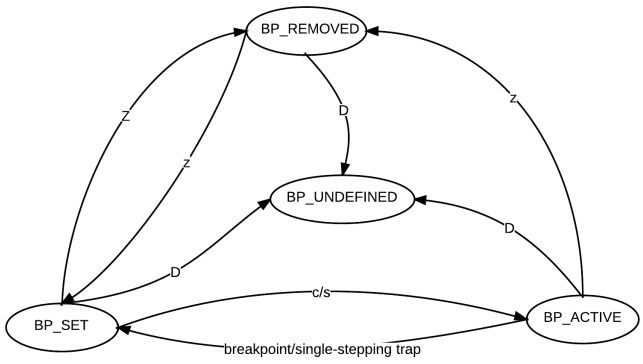

Figure 5: Breakpoint State Transitions

Similar to GDB's breakpoint implementation [9], we also choose to dynamically patch the instruction at the breakpoint address with a software breakpoint instruction, which is INT3 instruction in our case since Kitten is targeted for x86_64 architecture. INT3 instruction can replace the first byte of any x86_64 instruction, since it only accounts for 1 byte. Same as memory examination, we also need to do address translation before we replace the first byte of the instruction at breakpoint address. As a result of the replacement, whenever the processor encounters an INT3 instruction, it will stop the current debugging thread and transfer the current debugging thread's control and context to the GDB stub. Subsequently, the GDB stub will communicate the event to GDB client and take care of following GDB requests.

The GDB stub maintains a global breakpoint array for each debugging process. Each breakpoint is defined as a triplet consisting of a breakpoint address, a breakpoint state and the first byte of the replaced instruction. Similar to KGDB, each breakpoint has 4 possible states: BP_UNDEFINED, BP_SET, BP_ACTIVE, and BP_REMOVED. Figure 5 shows the breakpoint state transitions.

Initially, each entry of the breakpoint array is initialized to BP_UNDEFINED state. Upon receiving the D command to end the debugging session, the GDB stub restores all instructions replaced and set each entry of the breakpoint array back to BP_UNDEFINED state.

To set a breakpoint at a specific address, the GDB client sends Z command to the GDB stub, which subsequently simply saves the breakpoint address and sets the breakpoint state to BP_SET. However, breakpoints in BP_SET state are not active. The GDB stub will activate all breakpoints in BP_SET state once it receives a c/s command. It first saves the first byte of the instruction at breakpoint address, and then replaces the first byte of the instruction with INT3 instruction, and finally sets the breakpoint to BP_ACTIVE state. In contrast, all active breakpoints transit back to BP_SET state again when the current debugging thread hits a breakpoint or a single-stepping exception.

To remove a breakpoint at specific address upon receiving the z command, the GDB stub first restores the replaced instruction, and then sets the breakpoint to BP_REMOVED state. The introduction of BP_REMOVED state obviates the chance that the GDB stub has to handle a breakpoint exception already removed by another thread. As we mentioned before, if two threads t1 and t2 hit a breakpoint at the same time, only one of them (t1) is the focus of the debugging while the other one (t2) has to wait. So, there is a possibility that t1 removes the breakpoint hit by t2 while t2 is waiting. Later on, when t2 becomes debugging focus, the GDB stub will first check whether the breakpoint is removed before actually handling the exception.

3.6 Single-Stepping

To support single-stepping, we exploit the TRAP flag supported by x86_64 hardware. Upon receiving the s command, the GDB stub simply sets the TRAP flag of the current debugging thread. With the TRAP flag set, the thread will generate a single-stepping exception after each instruction it executes. Similar to breakpoint exception, the single-stepping exception will cause the control and context of the debugging thread transmitted to the GDB stub, which, in turn, will take responsibility of subsequent GDB requests.

4. CONCLUSIONS AND FUTURE WORK

In this paper, we describe the design and implementation of the GDB stub as a kernel module for Kitten without changing Kitten and Palacios architecture yet with minimal modification to Kitten and Palacios codebases. The GDB stub has only 590 lines of changes to Kitten and 238 lines of changes to Palacios in total. We are now able to debug a user-level application running in Kitten from a GDB client running in a Palacios Linux Guest, facilitating the application development, debugging and testing in Kitten. As part of the future work, we plan to add networking support for the GDB stub over Portals network programming API [13] for more efficient guest and host communication. Also, we want to extend the gdb stub to support MPI application debugging based on eclipse PTP debugger [14].

5. REFERENCES

[1] J. Lange, P. Dinda, K. Hale, and L. Xia, "An introduction to the palacios virtual machine monitor—version 1.3," 2011.

[2] J. Lange, K. Pedretti, T. Hudson, P. Dinda, Z. Cui, L. Xia, P. Bridges, A. Gocke, S. Jaconette, M. Levenhagen, *et al.*, "Palacios and kitten: New high performance operating systems for scalable virtualized and native supercomputing," in *Parallel and Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pp. 1–12, IEEE, 2010.

[3] P. Widener, S. Jaconette, P. G. Bridges, L. Xia, P. Dinda, Z. Cui, J. Lange, T. B. Hudson, M. J. Levenhagen, K. T. T. Pedretti, *et al.*, "Palacios and kitten: high performance operating systems for scalable virtualized and native supercomputing," tech. rep., Sandia National Laboratories, 2009.

[4] "Gdb: The gnu project debugger." `https://www.gnu.org/software/gdb/`.

[5] "Kgdb." `https://kgdb.wiki.kernel.org`.

[6] "Kgdb soucre code analysis(2.6.27)." `http://www.kgdb.info/kgdb/understand_kgdb/`

kgdb_source_2-6-27/.

[7] B. Gatliff, "Embedding with gnu: the gdb remote serial protocol," *Embedded Systems Programming*, vol. 12, pp. 108–113, 1999.

[8] B. Gatliff, "Embedding with gnu: Gnu debugger," *Embedded Systems Programming*, vol. 12, pp. 80–95, 1999.

[9] M. Tan, "A minimal gdb stub for embedded remote debugging.," *Columbia University*, 2002.

[10] J. Bennett, "Howto: Gdb remote serial protocol writing a rsp server," *Application Note 4*, 2008.

[11] "Gdb remote serial protocol." http://sourceware.org/gdb/onlinedocs/gdb/Remote-Protocol.html.

[12] N. Sidwell, V. Prus, P. Alves, S. Loosemore, and J. Blandy, "Non-stop multi-threaded debugging in gdb," in *GCC Developers' Summit*, p. 117, 2008.

[13] "Portals network programming interface." http://www.cs.sandia.gov/Portals/index.html.

[14] "Eclipse ptp scalable debugger manager." http://wiki.eclipse.org/PTP/designs/sdm.